Matrix GF2 Algebra
On A5/1 Explained III

By Juan Chamero, juan.chamero@intaq.org, as of April 2006

How to Get the Keyword once known one Initial State S(0)

Another A5/1 weakness provides a statistical attack shortcut

Registers R1, R2, and R3 shift at random, anyhow in a statistically regular fashion. For example when mixing 100 time steps in the average each one shifts 75 times and stops 25. If each Ri shifts around its mean with dispersion Sigma (i) we may select n(i) at random alternatives for each shift value having n^3 possible states along 100 steps for n=n(1)=n(2)=n(3). Not too much if we compute backwards along trees. The idea is to use matrices alternatively: State Transition Matrices Ci(j=>k) that transform State Si(j) in Sate Si(k) as follows:

$$Ci(j=>k)*Si(j) = Si(k), \text{ for } k>j, \text{ and conversely with its inverse } C'$$
$$C'(k=>j)*S(k) = Si(j)$$

Let's be then [72 73, 74, 75, 76, 77, 78] the spectrum interval of possible random shifts summa values [d1 d2 d3] for registers R1, R2, and R3. Being all shift counts equally probable we may go forward "via matrices" from a given state, for instance S(86), through all the 7^3 Cartesian Product shift triads to their corresponding 7^3 S(186) states. With a high probability one of these states will match the one that corresponds to that state S(86) running 100 time steps throughout the A5/1 algorithm. The same will apply for going backwards from a given S(186) state.

How such those matrices C look like?

They are rather simple structured. The basic shift operates in two "virtual steps", first shifts to its left, from lsb to mlb one position that represents a multiplication by 2 modulo 2^n, in this case 2^19, 2^22, and 2^23 for registers R1, R2, and R3 respectively. Second inserts as lsb the XOR of their corresponding Tab bits:

Warning: We say "virtual steps" because both operations, the shift and the feedback computation and insertion are simultaneous.

$$Si(j + 1) = Ci*Si(j)$$

With Ci stands for r(i)xr(i) matrices being r(i) their respective registers' range. In our A5/1 case C1 (19x19), C2 (22x22), and C3 (23x23). As tab bits are 18, 17, 16, and 13 for R1 its C1 matrix would be:

```
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   1   1
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   1   0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   0   1
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0   1   0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0   0   0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0   0   0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0   0   1
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0   1   1
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0   1   1
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0   1   0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0   0   0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0   0   1
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0   1   0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0   0   0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0   0   1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0   1   1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0   1   0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1   0   1
1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0   1   0
```

Note: Each row except last is a mask to catch a bit at a time virtually multiplying its "position value" by module 2^19

The powers of these matrices will traverse as many states as its power index. So Ci^3 applied to Si(11) will drive us to state Si(14). In the general case to jump k states starting from any known state could be performed by matrices. As we are working in GF2 and any number k could be expressed in this base, to create any C^k matrix we only need k components, namely:

k = 21 ⇔ 1 0 1 0 1 ⇔ C^21 = C^16*C^4*C^0

For our needs in this analysis (up to 100 state transformations) we only need to have pre computed the following basic module 2 matrices for each register:

C C^2 C^4 C^8 C^16 C^32 C^64

Enough computing math resources to cover jumps of up to 127 steps!. No problem with 2^n steps: the iterative process C <= C^2, executed n times.

Algorithm to Generate C basic powers

Define the tri-dimensional array C(i, j, h)
/in C(i, j, 0) we define the First State Transition Matrix/
/in C(i, j, h) we define the (h – 1) Transition Matrices/
h = 0, /h from 1 to H, maximum programmed jump (177) between states/
Do while h<=H, h++
i = 0
Do while i<=n, i++
j = 0
Do while j<=n, j++
C(i, j, h + 1) = 0
k=0
Do while k<n, k++
C(i, j, h + 1) = C(i, j, h+1) + C(i, k, h)*C(k, j, h)
/Optionally list below/
List all h planes
/List C matrices corresponding to powers: 2^0, 2^2, 2^2, 2^3, , …, 2^H/

DATA INPUT: First State Transition Matrices

C1(0): [010……..0], [001……..0], ……., [000……..1]; [1110010000000000000] being 19x19
C2(0): [010……..0], [001……..0],…….. [000…….. 1]; [11000000000000000000000] being 22x22
C3(0): [010……..0], [001……..0],…….. [000…….. 1]; [11100000000000010000000] being 23x23

Standard and non standard Shift Registers Logic

We have found the logic of our shift registers working at standard mode, meaning in two virtual steps to complete each iteration as follows:

Si(j + 1) = Ci*Si(j) that in fact implies two things: shift one position to the left, leaving the lsb of Register Ri “vacant”, to be filled with the Tap bits XORed enabling the whole pseudo random cycle. We are going to check all this reasoning for a 4 position shift register to see what we mean saying standard mode in order to introduce the algebra of a more complex mode as used in the first 86 steps of A5/1 algorithm.

n = 4
Cycle: 2^4 – 1 = 15
Seed: 0001

1	0001
2	0010
3	0100
4	1001
5	0011
6	0110
7	1101
8	1010
9	0101
10	1011
11	0111
12	1111
13	1110
14	1100
15	1000
16	0001
17	…….

The corresponding 4 “Characteristic” matrices are:

C^1				C^2				C^4				C^8			
0	1	0	0	0	0	1	0	1	1	0	0	1	0	1	0
0	0	1	0	0	0	0	1	0	1	1	0	0	1	0	1
0	0	0	1	1	1	0	0	0	0	1	1	1	1	1	0
21	1	0	0	0	1	1	0	1	1	0	1	0	1	1	1

Si(11) = (C^4)*Si(7) =>

C^4				Si(7)	Si(11)
1	1	0	0	1	0
0	1	1	0	1	1
0	0	1	1	0	1
1	1	0	1	1	1

Warning: In this algebra the multiplication of bits (represented either by (.) or (*)) corresponds to the “logic product” (AND) meanwhile the SUM corresponds to the XOR operator (represented either by (+) or same symbol within a circle). For both operators are valid the associative property:

C^k*(C^h*S(j) + K) = C^*(k + h)S(j) + C^k*K

As explained we may combine positional characteristic matrices in order to obtain any “jump” in distance between any pair of states. For example the operator C^11 could be obtained by multiplying C^8*C^2*C^1 giving the jump [1011] obtained by the corresponding truth values (C^8 (True), C^4(False), C^2(True), C^1(True)).

Unconventional Use of Shift Registers

A5/1 uses Shift Registers in several modes. In the beginning registers are XORed bit per bit, in parallel, from Kc first and then from Frames counter. This is equivalent to insert a random Tap bit with an unknown content that is equivalent to an exogenous intrusion. Let’s try to unveil the algebra of this mode.

We are going to compute “by hand” all the possible outputs when triggered by an arbitrary Kc, for example [0001]. We have to take into account that initially registers are zeroed.

Kc: 0001
S(0): 0000
S(1): 0001
S(2): 0010
S(3): 0100
S(4): 1001

Kc: 1101
S(0): 0000
S(1): 0001
S(2): 0010
S(3): 0101
S(4): 1010

Once understood the procedure XORing the designed tap bits with the bits from Kc, one at a time, we may generate the following correspondences:

Kc	S(4)
0001	1001
0010	0100
0100	0010
1001	1000
0011	1101
0110	0110
1101	1010
1010	0101
0101	1011
1011	1100
0111	1111
1111	1110
1110	0111
1100	0011
1000	0001

Recursively we may obtain this series performing the following recursion:

S(0) = 0
S(1) = C*0 + K1
S(2) = C*S(1) + K2
S(3) = C*S(2) + K3
S(4) = C*S(3) + K4

Where the Kj are n size vectors (for our example n = 4) with all zeroes except the lsb that is the Kc(j) bit, from right (lsb) to left. In our example:

K1: [0 0 0 Kc(1)] first bit from right side
K2: [0 0 0 Kc(2)] second bit from right side
K3: [0 0 0 Kc(3)] third bit from right side
K4: [0 0 0 Kc(4)] fourth bit from right side

S(4) = C*(C*S(2) + K3) + K4 =>
C*(C*(C*S(1) + K2) + K3) + K4 =>
C*(C*(C*(0 + K1) K2) + K3) + K4 =>
S(4) = C^3*K1 + C^2*K2 + C^1*K3 + K4

And in general for n = 64

S(64) = C^63*K1 + C^62*K2 + …….. + C^2*K62 + C^1*K63 + K64

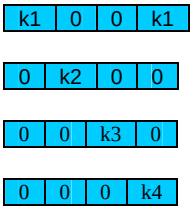
Let's now go back to our example:

S(4) = C^3*K1 + C^2*K2 + C^1*K3 + K4

Let us deep a little in these n products of a transition matrix by vectors that only have a meaningful component (0 or 1) in its last position. In fact for any C^(n-h)*Kh, for instance C^3K1, the n size vectors components result from the product of the coefficients of the last column by Kc(j). For our example we have:

C**3				K1		C**2				K2		C**1				K3		C**0				K4
0	0	0	1	0		0	0	1	0	0		0	1	0	0	0		1	0	0	0	0
1	1	0	0	0		0	0	0	1	0		0	0	1	0	0		0	1	0	0	0
0	1	1	0	0		1	1	0	0	0		0	0	0	1	0		0	0	1	0	0
0	0	1	1	k1		0	1	1	0	k2		1	1	0	0	k3		0	0	0	1	k4
k1	0	0	k1			0	k2	0	0			0	0	k3	0			0	0	0	k4	

We appreciate that product of matrices by keyword bits vectors K are reduced to a match between the last column of matrices that with their 1's signal where to insert the corresponding bit kj of Kc(j). The states are then computed as a result of XORing the following 4 (n) vectors:



And in the general case the XORing of n vectors of n bits each. Algebraically the linear combinations are as follows:

For n=4:

First Vector: [C3(1 4).k1, C3(2 4).k1, C3(3 4).k1, C3(4 4).k1]
Second Vector: [C2(1 4).k2, C2(2 4).k2, C2(3 4).k2, C2(4 4).k2]
Third Vector: [C1(1 4).k3, C1(2 4).k3, C1(3 4).k3, C1(4 4).k3]
Fourth Vector: [C0(1 4).k4, C0(2 4).k4, C0(3 4).k4, C0(4 4).k4]

Where the C0´s are all zeroes, except C0(n n) = 1. As we XOR them column by column we have the following expressions for state S(n):

S(4, 1) = C3(1 4).k1 + C2(1 4).k2 + C1(1 4).k3 + 0
S(4, 2) = C3(2 4).k1 + C2(2 4).k2 + C1(2 4).k3 + 0
S(4, 3) = C3(3 4).k1 + C2(3 4).k2 + C1(3 4).k3 + 0
S(4, 4) = C3(4 4).k1 + C2(4 4).k2 + C1(4 4).k3 + k4

And in general:

S(64, j) = C63(j 64).k1 + C62(j 64).k2 + C61(j 64).k3 + ……+ C1(j 64). k63 + k64

This could be synthesized by the expression:

$$S(n) = C64*K$$

Where C64 is the matrix formed by the last columns of the (n-1) matrices from C^1 to C^(n-1) with its right most column the unity vector [0 0 …….01], and K = Kc. Now knowing S(n) we may unveil K by inverting C64:

$$\mathbf{K = C64´*S(n)}$$

C64 and C64´ have 4096 coefficients but is a sparse matrix, plenty of zeroes.

Transition Matrices for A5/1 Algorithm

Our aim now is to show how this matrix approach works for states defined via concatenation of several Shift Registers complicated by mixing Tap bits XORing with exogenous bits, for instance entering in parallel, bitwise two strings corresponding to a private and a public key. We are going to see how it works in a simplified scheme of three small registers:

R1 [b3 b2 b1 b0];
R2 [b2 b1 b0];
R3 [b1 b0]
S(t): R1(t)OR2(t)OR3(t) ⇔ [b31(t) b21(t) b11(t) b01(t) b22(t) b12(t) b02(t) b13(t) b03(t)]

That is we may deal with registers as separate entities only linked to generate “output” bits by XORing bits coming out from most significant bits of three registers in the next cycle, namely:

$$y(t+1) = u1(t) + u2(t) + u3(t)$$

Where u1(t) = b31(t); u2(t) = b22(t); u3(t) = b13(t), are the Shift Registers outputs respectively.

Note: The (+) symbol refers to XOR binary operation that “adds” two members at a time, no matter the order because this operator is fully associative. For three members (u1, u2, u3) its equivalent “Truth Table” is as follows:

u1	u2	u3	u2+u3	u1+u2+u3	parity
0	0	0	0	0	0
0	0	1	1	1	1
0	1	0	1	1	1
0	1	1	0	0	0
1	0	0	0	1	1
1	1	0	1	0	0
1	1	1	0	1	1

We initiate the process zeroing registers as A5/1 does for each frame. We are going to analyze this sample for K = [1 0 1 1 0 1 1 0 1], generating transitions from S(0) to S(9) instead of transitions S(0) to S(64) in the A5/1 case using a private key K of 64 bits.

Note: As we will see later, “Initial States”, are those states S(86) after finishing stages one and two, traversing 64 + 22 states, 64 to mix registers with the private key and last 22 to mix registers with the 22 bits Frame counter respectively.

S(t)		R1				K			R2			K			R3		K
0		0	0	0	0	1			0	0	0	1			0	0	1
1		0	0	0	1	0			0	0	1	0			0	1	0
2		0	0	1	0	1			0	1	0	1			1	1	1
3		0	1	0	1	1			1	0	0	1			1	1	1
4		1	0	1	0	0			0	0	0	0			1	1	0
5		0	1	0	1	1			0	0	0	1			1	0	1
6		1	0	1	0	1			0	0	1	1			0	1	1
7		0	1	0	0	0			0	1	1	0			1	0	0
8		1	0	0	1	1			1	1	1	1			0	1	1
9		0	0	1	0				1	1	0				1	0	

States transitions from

S(0) = [0 0 0 0 0 0 0 0 0] to [0 0 1 0 1 1 0 1 0] via the Private Key
K = [1 0 1 1 0 1 1 0 1]

As we have demonstrated any partial state transition could be seen as the following iterative process:

Si(1) = Ci(1)*Si(0) + K1
Si(2) = Ci(1)*Si(1) + K2
……
Si(t) (3)= Ci(1)*Si(t-1) + Kt

Where Kj stands for a vector of size n (in this case n=9) with Kj(n) = K(n) and Kj(h) =0, for h=1, 2,…, (n-1). We use here the notation from 1 to n for bits because it looks easier for the iteration understanding. If (t) is limited to n steps as it is the case we have a sequence from S(0) to S(9). Really there is no limitation in t if we want to go farther because we may either stop entering bits of K or start to re-enter them cyclically. The Kj series looks like:

K1: [0 0 0 0 0 0 0 0 K(1)]
K2: [0 0 0 0 0 0 0 0 K(2)]
……
Kn: [0 0 0 0 0 0 0 0 K(n)]

Let’s deep a little on the nature of each iteration expression:

Note: Ci(0) are states transition matrices to traverse only one step, for instance from state 0 to state 1 and from state j to (j+1) state. Then we have a product of a nxn C matrix by a n size S vector and then XORing the result with vectors Kj.

If we want to jump directly from S(0) to S(n) we have:

$$Si(9) = Ci(1)^8*Si(0) + Ci(1)^8*K1 + Ci(1)^7*K2 + ... + Ci(1)^1*K8 + K9$$

If we make S(0) = 0 the first term disappear. The mechanic of this process is very systematic because the Ci(h) matrices cycle throughout (2^ri – 1) different types, being ri the order of the registers, 19, 22 and 23 in the A5/1 case, and 4, 3, and 2 in our small example. This allow us to jump from any non zero state j to any state (j+h) throughout Ci(1)^h matrices. To make things easy we use the following notation:

$$Ci(1)^h = Ci(h)$$

For our example we have then: Ci(1), Ci(2), Ci(3), ……, Ci(2^n – 1), being these last matrices shifted Unity matrices because they start new cycles. Let's see the series for our example:

C1 Matrices

C1(1)			
0	1	0	0
0	0	1	0
0	0	0	1
1	1	0	0

C1(2)			
0	0	1	0
0	0	0	1
1	1	0	0
0	1	1	0

C1(3)			
0	0	0	1
1	0	0	0
0	1	1	0
0	0	1	1

C1(4)			
1	1	0	0
0	1	1	0
0	0	1	1
1	1	0	1

C1(5)			
0	1	1	0
0	0	1	1
1	1	0	1
1	0	1	0

C1(6)			
0	0	1	1
1	1	0	1
1	0	1	0
0	1	0	1

C1(7)			
1	1	0	1
1	0	1	0
0	1	0	1
1	1	1	0

C1(8)			
1	0	1	0
0	1	0	1
1	1	1	0
0	1	1	1

C1(15)			
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Note: we have omitted matrices C1(9) to C1(14).

C2 Matrices

C2(1)		
0	1	0
0	0	1
1	1	0

C2(2)		
0	0	1
1	1	0
0	1	1

C2(3)		
1	1	0
0	1	1
1	1	1

C2(4)		
0	1	1
1	1	1
1	0	1

C2(5)		
1	1	1
1	0	1
1	0	0

C2(6)		
1	0	1
1	0	0
0	1	0

C2(7)		
1	0	0
0	1	0
0	0	1

C3 Matrices

C3(1)	
0	1
1	1

C3(2)	
1	1
1	0

C3(3)	
1	0
0	1